WEEK 3

Q – Repeated String Match

```
class Solution {
public:
    int repeatedStringMatch(string a, string b) {
         string s = "";
          int count = 0;
          while (s.size() < b.size())
          {
              s += a;
              count++;
          }
          if (s.find(b) != string::npos)
              return count;
          s += a;
          count++;
          if (s.find(b) != string::npos)
              return count;
              return -1;
    }
};
```

Q- Implement Strstr

```
class Solution {
public:
    int strStr(string haystack, string needle) {
        if(needle.size()>haystack.size())
    {
```

```
        return -1;
    }
    for(int i=0;i<haystack.size();i++)
    {
        int j=0;
        for(j=0;j<needle.size();j++)
        {
            if(haystack[j+i]!=needle[j])
                break;
        }
        if(needle.size()==j)
            return i;
    }
    return -1;
    }
};
```

Q – Minimum Characters required to make a string Palindrome

```
int Solution::solve(string A) {
    int n = A.length();
    int left = 0;
    int right = n - 1;
    int added = 0;
    while (left < right) {

        if (A[left] == A[right]) {
            left++;
            right--;
        }
```

```
        else {
          left = 0;
          added = n - right;
          while (A[left] == A[right]) {
            added--;
            left++;
          }
          right--;
        }
    }
    return added;
}
```

Q- Check for Anagrams

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        map<char,int> m;

                        for(auto i:s)
                        {
                                m[i]++;
                        }
                        for(auto i:t)
                        {
                                if(m.find(i)==m.end() || m[i]==0) return false;
                                else m[i]--;
                        }
```

```
            for(auto i:m)
            {
                if(i.second>0) return false;
            }
            return true;
    }
};
```

Q- Count And Say

```
class Solution {
public:
    string countAndSay(int n) {
        if (n == 1)     return "1";
    if (n == 2)     return "11";

    string str = "11";
    for (int i = 3; i<=n; i++)
    {
        str += '$';
        int len = str.length();

        int cnt = 1;
        string  tmp = "";

        for (int j = 1; j < len; j++)
        {
            if (str[j] != str[j-1])
            {
                tmp += to_string(cnt) ;
```

```
                tmp += str[j-1];

                cnt = 1;
            }
            else cnt++;
        }
        str = tmp;
    }
return str;
    }
};
```

Q- Compare Version Numbers

```
class Solution {
public:
    vector<string>compute(string s){
      vector<string>a;
        int index = 0;

        int i = 0;
        string tmp = "";
        while(i < s.size())
        {
            if(s[i] == '.'){
                a.push_back(tmp);
                tmp = "";
                i++;
```

```
        }
        else if(s[i] == '0' && tmp == "")i++;
        else tmp.push_back(s[i++]);
    }
    a.push_back(tmp);
    return a;
}

bool comp(string a , string b)
{
    if(a.size() != b.size())
        return a.size() > b.size();
    return a > b;
}

int compareVersion(string version1, string version2) {
    vector<string>a = compute(version1);
    vector<string>b = compute(version2);

    while(a.size() < b.size())a.push_back("");
    while(b.size() < a.size())b.push_back("");

    for(int i=0;i<a.size();i++){


        if(comp(a[i] , b[i])) return 1;
        else if(comp(b[i] , a[i])) return -1;
    }
    return 0;
}
```

```
};
```

Q- Maximum Width of Binary Tree

```
int widthOfBinaryTree(node * root) {
  if (!root)
    return 0;
  int ans = 0;
  queue < pair < node * , int >> q;
  q.push({
    root,
    0
  });
  while (!q.empty()) {
    int size = q.size();
    int curMin = q.front().second;
    int leftMost, rightMost;
    for (int i = 0; i < size; i++) {
      int cur_id = q.front().second - curMin;
      node * temp = q.front().first;
      q.pop();
      if (i == 0) leftMost = cur_id;
      if (i == size - 1) rightMost = cur_id;
      if (temp -> left)
        q.push({
          temp -> left,
          cur_id * 2 + 1
        });
      if (temp -> right)
```

```
        q.push({
          temp -> right,
          cur_id * 2 + 2
        });
    }
    ans = max(ans, rightMost - leftMost + 1);
  }
  return ans;
}
```

Q- Level Order Traversal

```
class Solution {
public:
    vector<int> levelOrder(TreeNode* root) {
        vector<int> ans;
        if(root == NULL)
            return ans;
        queue<TreeNode*> q;
        q.push(root);
        while(!q.empty()) {

            TreeNode *temp = q.front();
            q.pop();

            if(temp->left != NULL)
                q.push(temp->left);
            if(temp->right != NULL)
                q.push(temp->right);
```

```
            ans.push_back(temp->val);
        }
        return ans;
    }
};
```

Q- Height of Binary Tree

```
class Solution {
 public: int maxDepth(TreeNode* root) {
 if(root == NULL) return 0;
 int lh = maxDepth(root->left);
 int rh = maxDepth(root->right);
 return 1 + max(lh, rh);
 } };
```

Q- Calculate the Diameter of a Binary Tree

```
class Solution {
public:
    int diameterOfBinaryTree(TreeNode* root) {
        int diameter = 0;
        height(root, diameter);
        return diameter;
    }

private:
    int height(TreeNode* node, int& diameter) {

        if (!node) {
            return 0;
```

```
        }

        int lh = height(node->left, diameter);
        int rh = height(node->right, diameter);

        diameter = max(diameter, lh + rh);

        return 1 + max(lh, rh);
    }
};
```

Q- Check if the binary tree is balanced

```
class Solution {

public:
    bool isBalanced(TreeNode* root) {
        return dfsHeight (root) != -1;
    }

    int dfsHeight (TreeNode *root) {

        if (root == NULL) return 0;

        int leftHeight = dfsHeight (root -> left);

        if (leftHeight == -1)
            return -1;

        int rightHeight = dfsHeight (root -> right);
```

```
        if (rightHeight == -1)
          return -1;

        if (abs(leftHeight - rightHeight) > 1)
          return -1;

        return max (leftHeight, rightHeight) + 1;
    }
};
```

Q- Symmetric Binary Tree

```
bool isSymmetricUtil(node * root1, node * root2) {
  if (root1 == NULL || root2 == NULL)
    return root1 == root2;
  return (root1 -> data == root2 -> data) && isSymmetricUtil(root1 -> left, root2->
  right) && isSymmetricUtil(root1 -> right, root2 -> left);
}
bool isSymmetric(node * root) {
  if (!root) return true;
  return isSymmetricUtil(root -> left, root -> right);
}
```

Q- Flatten Binary Tree to Linked List

```
void flatten(node* root) {
        node* cur = root;
                    while (cur)
                    {
                            if(cur->left)
```

```
                {
                        node* pre = cur->left;
                        while(pre->right)
                        {
                                pre = pre->right;
                        }
                        pre->right = cur->right;
                        cur->right = cur->left;
                        cur->left = NULL;
                }
                cur = cur->right;
        }
  }
```

Q- Populate next right pointers of tree

```
 void util(Node* root, map<int,vector<Node*>> &mp,int level)
  {
     if(root==NULL)
     {
        return;
     }

     if(mp.find(level)!=mp.end())
     {
        vector<Node*> temp=mp[level];
        int n=temp.size();
        Node* prev=temp[n-1];
```

```
        prev->next=root;
        mp[level].push_back(root);
    }

    else
    {
        mp[level].push_back(root);
    }
    util(root->left,mp,level+1);
    util(root->right,mp,level+1);
}
Node* connect(Node* root) {
    map<int,vector<Node*>> mp;
    util(root,mp,1);

    for(auto it:mp)
    {
        vector<Node*> temp=it.second;
        int n=temp.size()-1;
        Node* last=temp[n];
        last->next=NULL;
    }
    return root;
}
```